Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Uroš Zaletelj

Razvoj domensko specifičnega jezika za generiranje Mocha testov

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: viš. pred. dr. Igor Rožanc

Ljubljana, 2017

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:
Razvoj domensko specifičnega jezika za generiranje Mocha testov

Tematika naloge:

Domensko-specifični jeziki (DSL) so programski jeziki, ki so namenjeni učinkovitemu reševanju problemov v izbrani problemski domeni. Kot taki so bliže razmišljanju uporabnika in mu lahko precej olajšajo delo.

V diplomski nalogi najprej preučite in predstavite področje razvoja DSL-jev. Na podlagi tega zasnujte in izdelajte notranji DSL za poenostavljeno pisanje testov v okolju Node.js, ki omogoča tvorbo testov v testnem ogrodju Mocha. Njegovo uporabo prikažite na primeru testiranja spletne aplikacije za najem avtomobilov. Nalogo zaključite s primerjalno analizo novega načina testiranja s predhodnim načinom.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Uroš Zaletelj sem avtor diplomskega dela z naslovom:

Razvoj domensko specifičnega jezika za generiranje Mocha testov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom viš. pred. dr. Igorja Rožanca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva

V Ljubljani, dne 14. julija 2017 Podpis avtorja:

Hvala mentorju viš. pred. dr. Igorju Rožancu za pomoč pri izdelavi diplomskega dela. Hvala družini za vso podporo med študijem.

Družini

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
DSL	Domain-Specific Language	domensko-specifični jezik
GPL	General-purpose language	splošnonamenski jezik
HTML	HyperText Markup Language	jezik za označevanje nadbesedila
URL	Uniform Resource Locator	enotni naslov vira
npm	Node.js Package Manager	urejevalnik paketov za okolje Node.js

Povzetek

Cilj diplomskega dela je razvoj ter implementacija domensko-specifičnega jezika (DSL-ja), ki omogoča hitrejše ter predvsem lažje pisanje Mocha testov. DSL-ji so običajno manjši programski jeziki, ki se večinoma osredotočajo na probleme v posamezni aplikacijski domeni. Pri razvoju rešitve bomo uporabili okolje Node.js, za programski jezik javascript. Za testno okolje bomo uporabili okolje Mocha, ki je najpogosteje uporabljeno okolje za programski jezik javascript.

Glavne zahteve pri razvoju domensko-specifičnega jezika so, da testiranje poenostavimo, odstranimo vso nepotrebno kodo ter izboljšamo sam izpis rezultatov.V uvodnem poglavju na kratko opišemo tehnologije, za katere so namenjeni testi, predstavimo problem, ter na kratko opišemo pričakovanja po končani diplomski nalogi. Drugo poglavje predstavi značilnosti DSL-jev. Izpostavimo predvsem njihove lastnosti ter delitev v skupine. Med prednosti sodi predvsem prilagojenost dejanskim uporabnikom, za nas pa je še najbolj pomembna skupina notranjih DSL-jev. Na koncu predstavimo še tehnologije, ki so bile uporabljene pri razvoju jezika. V tretjem poglavju podrobneje opišemo izvorni problem ter razložimo, zakaj smo izbrali določene tehnologije, kaj so njihove slabosti ter predvsem prednosti. Node.js je najpreprostejšo okolje za implementacijo notranjega DSL-ja, ker ga razvijalci - testerji že dobro poznajo. V tem poglavju so predstavljeni tudi osnovni gradniki DSL-ja. V četrtem poglavju prikažemo uporabo ter delovanje jezika pri implementaciji testov na realnem projektu. Na kratko opišemo projekt ter predstavimo potek testiranja od začetka pisanja testov pa vse do izpisa

rezultatov testiranja. V zadnjem poglavju predstavimo sklepne ugotovitve. Izpostavimo predvsem dejstvo, da je DSL veliko učinkovitejši pri pisanju testov, obstajajo pa seveda možnosti nadaljnega razvoja jezika.

Ključne besede: testiranje, Mocha, domensko-specifični jeziki, Node.js.

Abstract

The aim of the diploma thesis is to present development and implementation of domain-specific language that enables a faster and mostly easier writing of Mocha tests. Domain-specific languages are minor computer languages that mainly focus on problems in a particular application domain. For developing a solution, the Node.js environment for JavaScript computer language is used. Our testing environment will be Mocha, the most frequently used environment for JavaScript platform. A tester will use a simplified solution in Mocha environment.

The main requirements for domain-specific language development are the simplified testing as we eliminate unnecessary code and improve the printing of results.

In the Introduction, we will briefly describe technologies for which tests are meant, present a problem that needs to be solved, and expectations after the finished thesis.

The second chapter will introduce characteristics of domain-specific languages. We expose basic characteristics and we divide them into groups. Moreover, we present technologies used for the development of DSL.

The third chapter describes the primary problem and explains why certain technologies were used, as well as their advantages in disadvantages. Furthermore, we will present basic cornerstones required for construction of DSL.

The fourth chapter presents usage and activities of our language on a real project. We briefly describe the project and the process of testing.

In the last chapter we present final conclusions, emphasizing the fact the newly created language is much more efficient at test writing. We present the possibilities of further development of the language as well.

Keywords: testing, Mocha, domain-specific languages, Node.js.

Poglavje 1

Uvod

Pri razvoju zalednega sistema je ena najpomembnejših stvari ta, da celoten sistem pred oddajo naročniku čimbolj testiramo. Testiranje tovrstne rešitve je možno, saj za vsak zahtevek točno vemo, kakšen mora biti pričakovan odgovor. Pri izbiri testnega okolja smo omejeni na infrastrukturo ter tehnologije, ki jih uporabljamo pri razvoju zalednega sistema. To pomeni, da moramo poiskati testno okolje, ki je namenjeno uporabljeni tehnologiji, saj vsa testna okolja ne podpirajo vseh programskih jezikov.

V našem primeru je zaledni sistem napisan v jeziku *javascript* [1], kot okolje pa se uporablja *Node.js* [2]. Za testiranje se tako uporablja okolje *Mocha* [3], ki velja za najpogosteje uporabljeno testno okolje za programski jezik javascript.

Cilj tega diplomskega dela je razviti notranji domensko-specifični jezik (angl. domain-specific languages ali DSL) [4], ki bo še bolj poenostavil pisanje testov v tej tehnologiji. DSL-ji nam omogačajo lažje reševanje problema na točno določeni domeni. Kot primer DSL-ja lahko izpostavimo *HTML* [5], ki je namenjen izključno izdelovanju spletnih strani ali pa jezik *LaTeX* [6], katerega se uporablja za pripravljanje dokumentov. Mocha okolje bi lahko predstavili kot nekakšen domensko-specifični jezik, saj se osredotoči na nek ožji problem v našem primeru testiranje. Pri razvoju DSL-ja bo pomembno, koliko časa ter znanja bo potrebno za razvoj in kasneje učenje novega jezika.

DSL-je je mogoče razviti na različne načine. Mernik [7] je opisal proces izbire ter razvoja različnih DSL-jev, pri čemer je razvoj (zunanjega) DSL-ja lahko enako zahteven kot razvoj novega programskega jezika. Lahko pa uberemo tudi možnost in DSL razvijemo bolj enostavno in manj formalno. Mi bomo uporabili neformalni pristop pri razvoju DSL-ja, ki ga bomo zasnovali agilno, saj želimo razvoj res poenostaviti jezik pa čimbolj približati željam testerjev. Končen cilj bo pokazati upravičenost, torej, da z uporabo DSL-ja pridobimo na hitrosti ter enostavnosti pisanja testov.

Glavni del diplomskega dela bo tako prikaz razvoja DSL-ja za pisanje testov. Kot primer uporabe bomo nastali domensko-specifični jezik preizkusili na realnem projektu. Gre za aktiven projekt za zaledni sistem namenjen izposojevanju ter deljenju avtomobilomv med večjo skupino uporabnikov, na katerem trenutno sodeluje 5 razvijalcev, ki so poleg tega tudi testerji. Tudi avtor diplomskega dela sodeluje kot razvijalec na projektu. Na koncu bomo diplomsko delo zaključili z analizo uporabe jezika: primerjali bomo čas izvajanja, velikost napisane kode, pravilno izvedbo ter odziv uporabnikov. Tako bomo preverili, če je le-ta primeren za uporabo v praksi ali ne.

Poglavje 2

Domensko-specifični jeziki

2.1 Domensko-specifični jeziki

Definicija DSL-jev je zelo preprosta: DSL je jezik, ki je namenjen reševanju določenega problema v izbrani domeni [8]. V nadaljevanju se bomo najprej osredotočili na razliko med domensko-specifičnimi ter splošnonamenskimi jeziki (angl. general-purpose language - GPL) [9].

Programski jezik je stroju berljiv umetni jezik, ki je bil razvit, da izraža izračune, katere lahko izvaja stroj oziroma računalnik [10]. Med programskimi jeziki so najbolj razširjeni splošnonamenski programski jeziki. To so programski jeziki, ki so namenjeni razvoju programskih rešitev v različnih domenah ter ustrezajo Turingovemu testu. Programski jezik Turingovemu testu ustreza takrat, ko lahko s programskim jezikom rešimo vsak problem, ki se ga da izračunati tudi z Turingovim strojem [11]. To po drugi strani pomeni tudi to, da lahko vsak programski jezik brez težav zamenjamo z drugim. Domensko-specifični jeziki so običajno manjši programski jeziki, ki se večinoma osredotočajo na manjše probleme v posamezni aplikacijski domeni. Z domensko-specifičnimi jeziki ne moremo sprogramirati rešitve v poljubni domeni, lahko pa z njihovo uporabo lažje rešimo probleme v ustrezni domeni [12]. Meja med splošnonamenskimi jeziki in domensko-specifičnimi jeziki ni vedno jasno začrtana. Jezik je recimo lahko namenjen posamezni domeni,

Primerjava jezikov		
	Splošnonamenski jeziki	Domensko-specifični jeziki
Domena	velika	majhna
Obširnost jezika	velika	majhna
Hitrost razvoja	počasen	hiter
Učenje	obsežno	manjše
Zapadlost koode	skoraj nemogoča	možna glede na domeno
razvijalci	veliko	domenski eksperti

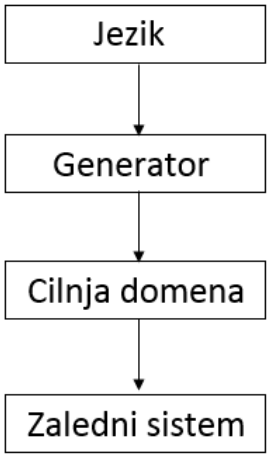
Tabela 2.1: Primerjava programskih jezikov

vendar se ga potem uporablja tudi v drugih domenah (recimo programski jezik javascript, kateri je v osnovi namenjen spletnim stranem uporabimo pa ga lahko tudi na zalednem sistemu). Po drugi strani pa je lahko jezik namenjen splošni uporabi, a je pretežno uporabljen le v posamezni domeni [4]. Tabela 2.1 predstavlja nekatere osnovne razlike med splošnonamenskimi ter DSL-ji.

Za primer splošnonamenskega jezika lahko vzamemo programski jezik java [13]. Java se uporablja tako v zalednih sistemih kot v spletnih brskalnikih in ima zelo široko bazo uporabnikov. Sam jezik je zelo obsežen ter se nenehno razvija.

Na drugi strani lahko za primer domensko-specifičnega jezika vzamemo jezik HTML, ki je namenjen izključno izdelovanju spletnih strani. HTML kljub temu vsebuje kar nekaj lastnosti splošnonamenskih jezikov: obširnost jezika je večja, učenje je daljše in zahtevnejše itd.

Pri razvoju domensko-specifičnega jezika moramo najprej preveriti, kako jezik deluje od modeliranja do izvedbe. Običajno govorimo o tronivojski arhitekturi: jezik, generator, ciljna domena (slika 2.1) [14].



Slika 2.1: Zasnova DSL [14]

Jezik je naš domensko-specifični jezik. Predstavlja abstraktni mehanizem, s katerim sprogramiramo ustrezno rešitev v dani domeni. Jezik je sestavljen iz konkretne sintakse (notacije za pisanje programov), abstraktne sintakse (podatkovnih modelov v tem jeziku) ter semantike (podatkovnih tipov in omejitev). Definiran je kot metamodel s pripadajočimi zapisi in orodji [14]. Kot primer lahko izpostavimo kar naš DSL.

Generator določa, kako se iz modelov pridobljene informacije pretvorijo v kodo. V najpreprostejših primerih vsak simbol predstavlja določen končni del kode. Pri izdelavi delujočega domensko-specifičnega jezika je cilj, da po izvedbi programa ni potrebno vložiti dodatnega dela do delujoče rešitve. S tega vidika poznamo generatorje, ki kodo DSL-ja pretvorijo v drugojezično kodo, ki se potem izvaja na ciljni domeni, ter na generatorje, ki kodo DSL-ja naložijo in neposredno poganjajo na ciljni domeni [14]. V naši konkretni rešitvi govorimo o prvi rešitvi, saj se naš DSL pretvori v ustrezno obliko za nadaljno uporabo.

Ciljna domena zagotavlja vmesnik med ustvarjeno kodo in ciljno platformo. V večini primerov je ciljna platforma vnaprej določena, zato se jezik ter generator prilagajata njej. [14]. V našem primer gre za okolje Node.js.

2.2 Razdelitev vlog pri razvoju DSL-ja

- *Domenski uporabniki.* Domenski uporabniki so v našem primeru testerji, ki uporabljajo novonastali DSL ter pišejo teste za ciljno domeno.
- *Domenski eksperti.* Domenski eksperti so razvijalci, ki uporavljajo testno okolje Mocha. Tako nam lahko ob razvoju jezika razložijo glavne lastnosti Mocha okolja.
- *Razvijalci jezika.* Razvijalec jezika je v našem primeru avtor diplomskega dela.

Omenimo naj, da v našem primeru avtor diplomskega dejansko predstavlja vse vloge pri razvoju jezika.

2.3 Delitev domensko-specifičnih jezikov

Domensko-specifične jezike v grobem delimo na **notranje** ter **zunanje** [15]. Obstajajo tudi druge, za nas manj pomembne delitve.

Notranji domensko-specifični jeziki so vgrajeni neposredno v okolje in se tako izvajajo v jeziku, ki ga že uporabljamo. Za primer lahko vzamemo okolje Mocha, ki se izvaja v programskem jeziku javascript. Ključno je, da Mocha lahko uporablja vse funkcije, ki jih omogoča javascript.

Pri notranjem domensko-specifičnem jeziku lahko uporabljamo vse lastnosti in operacije osnovnega jezika, zato včasih težko opazimo razliko med obema jezikoma. Slaba lastnost teh jezikov je, da so omejeni na sintakso osnovnega jezika [15].

Zunanji domensko-specifični jeziki so ločeni od glavnega programskega jezika, ki ga uporabljamo v cilni domeni. To pomeni, da se celotna

rešitev izvaja v drugem jeziku kot naša osnovna aplikacija. Zato za procesiranje zunanjega DSL-ja največkrat uporabimo razčlenjevalnik v ciljni domeni, da le-ta pretvori kodo v skupni jezik. Najpomembnejša prednost teh jezikov je, da lahko sintakso zelo dobro prilagodimo svojim željam ter tako domenskim ekspertom poenostavimo programiranje [15].

Od več dejavnikov je odvisno, za kateri jezik se na koncu odločimo. V večini primerov je rešitev možno implementirati tako z notranjim kot zunanjim domensko-specifičnim jezikom. Zato moramo pretehtati tako prednosti kot slabosti posamezne implementacije ter se na koncu odločiti za tisto, ki nam bolj ustreza. V nadaljevanju si bomo ogledali podoben primer odločanja med dvema rešitvama, ki delujeta na principu notranjih domensko-specifičnih jezikih.

2.4 Prednosti domensko-specifičnih jezikov

Domensko-specifični jeziki imajo v primerjavi s splošnonamenskimi jeziki naslednje prednosti: [11, 4, 14]

- **Učinkovitost**. Učinkovitost se poveča, ker se jezik osredotoči na točno določen problem v izbrani domeni. To se izkaže predvsem zato, ker ni potrebno pisati velike količine kode. Domensko-specifični jezik namreč poskrbi, da se napisana koda pretvori v ustreznejšo obliko, zato je koda lažje berljiva, zanjo pa porabimo tudi manj časa.

- **Kakovost**. Rezultat odstranitve nepotrebnih funkcij je tudi kakovostnejša koda, saj pri pisanju kode pride do manj napak. Če se lete pojavijo, jih običajno lažje odkrijemo. Koda je prav tako lažja za vzdrževanje, če je seveda jezik pravilno zasnovan.

- **Validacija in verifikacija**. Domensko-specifični jeziki so v večini primerov bolj semantični kot splošno-namenski jeziki, kar pomeni, da je jezik v tem primeru bolj odporen na napake, saj je točno določeno,

kakšne vrste podatke lahko podamo neki funkciji. V primeru, da je podatek napačen pa uporabnika o temu obvestimo [16]. To posledično pomeni, da je analiza ter pravilna uporaba jezika lažja, poleg tega pa jezik vrača razumljivejše napake.

- **Sodelovanje z domenskimi eksperti**. Domensko-specifični jeziki so v večini primerov razviti ter modelirani s pomočjo domenskih ekspertov, ki sodelujejo s programerji. To privede do tega, da programerji bolje spoznajo ciljno domeno, eksperti pa programiranje, zato se kasneje lažje lotijo programiranja v DSL-ju. V primeru, ko domenski eksperti ne programirajo, pa to povzroči, da lahko bolje sodelujejo pri validaciji končnega produkta. V najslabšem primeru ustvarimo vsaj ekspertu bolj priročna poročila ali simulacije.

- **Učinkovita orodja**. Zunanji domensko-specifični jeziki lahko uporabljajo razvojna okolja, ki se zavedajo posebnosti jezika. Rezultat tega je izboljšana uporabniška izkušnja, ki jo zagotovijo predvsem orodja kot so recimo avtomatsko zaključevanje posameznih ukazov ali razhroščevalnik. Vsa orodja pohitrijo in poenostavijo pisanje kode, zato izboljšajo učinkovitost.

- **Neodvisnost od ciljne platforme**. Domensko-specifični jeziki so v večini primerov implementirani tako, da imajo vmesen generator oziroma interpreter do cilnje platforme. To v praksi pomeni, da so popolnoma neodvisni in omogočajo spremembo interpeterja ali ciljne platforme.

- **Čas izvajanja**. Pri domensko-specifičnih jezikih, ki so namenjeni predvsem generiranju kode za ciljno platformo, se čas prevajanja kode do končne rešitve ne poveča, saj generator zgenerira kodo, ki je že optimizirana. To pomeni, da je čas izvajanja isti, vendar je programiranje z uporabo DSL-ja hitrejše.

2.5 Slabosti domensko-specifičnih jezikov

Domensko-specifični jeziki imajo seveda tudi nekaj slabosti: [11, 4, 14]

- **Čas razvoja jezika**. Pri razvoju novega jezika se je treba držati ustreznega postopka. V [7] je v ta namen predstavljen proces razvoja DSL-ja. To ni enostavna naloga in za razvoj porabimo kar nekaj časa, ki bi ga lahko namenili drugim nalogam v okviru izdelave ciljne rešitve.

- **Učenje jezika**. Ko razvijemo nov jezik, se ga je potrebno naučiti. Domenski eksperti se tako seznanijo predvsem s funkcijami, ki jih jezik omogoča, ter s prijemi za programiranje. Za to je potrebno kar nekaj časa, saj ponavadi nimajo znanja programiranja.

- **Vzdrževanje**. Po končani izdelavi jezika je potrebno jezik vzdrževati ter ga prilagajati novim spremembam. V večini primerov so spremembe nujne, saj sicer čez čas jezik ob spremembi ciljne domene preneha ustrezno delovati.

- **Odvisnost od domensko-specifičnih jezikov**. Pri razvoju celovite rešitve lahko zelo hitro uporabimo več domensko-specifičnih jezikov. To sicer pomeni modularno rešitev, kjer razvoj poteka hitreje. Po drugi strani pa se začnemo preveč zanašati na jezike, ki smo jih uporabili. Ob morebitni spremembi se tako lahko znajdemo v težavah, ko moramo posodabljati že napisano kodo.

- **Kvaliteta izdelave**. Za kvalitetno izdelavo domensko-specifičnega jezika potrebujemo veliko časa ter izkušenj. Ob morebitni slabši implementaciji jezika si lahko zelo otežimo nadaljne delo. V tem primeru nam jezik postane bolj v breme kot pa v prednost.

2.6 Tehnologije

2.6.1 Javascript

Javascript je objektni skriptni programski jezik, ki ga je razvil Netscape, da bi spletnim programerjem pomagal pri ustvarjanju interaktivnih spletnih strani. Jezik je bil razvit neodvisno od Jave, vendar si z njo deli številne lastnosti in strukturo. JavaScript lahko sodeluje s HTML kodo in ji doda dinamično izvajanje. JavaScript podpirajo velika programska podjetja in ga kot odprt jezik lahko uporablja vsakdo, ne da bi zato potreboval licenco. Podpirajo ga vsi novejši spletni brskalniki. Zadnja stabilna različica pa je uradno standardiziran ECMAScript 6 [1]. Primer kode je prikazan na kodi 2.1.

```
1 var a = ["a", "b", "c"];
2 a.forEach(function(entry) {
3     console.log(entry);
4 });
```

Koda 2.1: Primer Javascript kode

2.6.2 Node.js

Za razvoj našega domensko-specifičnega jezika smo izbrali okolje Node.js [2], ki za pisanje kode uporablja programski jezik JavaScript[1]. To odločitev bomo podrobneje predstavili v nadaljevanju, tu pa želimo na kratko predstaviti samo okolje Node.js.

Node.js je okolje za *JavaScript*, ki je zgrajeno na Chrome V8 Javascript interpreterju. Temelji na dogodkovnem modelu, ki celotno okolje naredi nezahtevno ter zelo učinkovito [2]. Ker celoten sistem temelji na asinhronem dogodkovnem JavaScript pogonu, je Node.js namenjen za izgradnjo razširljivih spletnih aplikacij. Sistem omogoča, da je lahko nanj v nekem trenutku povezanih več aktivnih povezav, ki jih sistem analizira, izvede zahtevane aktivnosti ter vrne ustrezen odgovor. Samo okolje omogoča tudi uporabo velikega

števila vnaprej pripravljenih knjižnic - paketov (angl. *packages*), ki so jih pripravili razvijalci za širšo uporabo. Do paketov lahko dostopamo preko *npm* paketnega sistema (angl. *Node.js package manager - npm*), ki nam omogoča, da pakete hitro in enostavno dodamo v svoj projekt [17]. Na sliki 2.2 je prikazan logotip Node.js.

Slika 2.2: Logotip Node.js

Za Node.js smo se odločili predvsem zaradi njegove enostavne uporabe Prav tako pa se to okolje že uporablja na ciljni platformi, zato imamo že veliko potrebnega znanja za razvoj jezika. Tudi učenje novonastalega jezika bo potekalo hitreje kot v primeru drugih alternativ, ki jih bomo predstavili v nadaljevanju.

2.6.3 Mocha

Mocha je testno okolje za JavaScript, ki se izvaja v Node.js okolju [3]. Deluje tudi v vseh spletnih brskalnikih, zato lahko zelo enostavno testiramo tudi asinhrono izvajanje kode. Mocha testi se izvajajo zaporedno, kar omogoča prilagodljivo in natančno testiranje [3]. Na sliki 2.3 je prikazan Mocha logotip.

Na naši ciljni platformi se za testiranje uporablja Mocha okolje. Okolje ponuja zelo obsežen nabor funkcij, veliko tudi takšnih, ki jih v našem primeru ne potrebujemo. Zato bo naš domnesko-specifični jezik podpiral le potrebni del funkcionalnosti. Naj omenimo, da se bodo testi v ozadju še vedno poganjali preko tega okolja, naš jezik pa bo le generator, ki bo odstra-

Slika 2.3: Logotip Mocha

nil nepotrebno kodo. Tako bo tester imel enostavnejše delo pri pisanju testov.

```
describe('Countries', function() {
    it('should return Slovenia', (done) => {
      request(app)
        .get(`${host}?filters={"code":["SI"]}`)
        .set('Accept', 'application/json')
        .end((err, res) => {
          assert.equal(res.body[0].code, 'SI');
          assert.equal(res.body[0].name, 'Slovenia');
          done();
        });
    });
});
```

Koda 2.2: Primer Mocha testa

V primeru kode 2.2 lahko vidimo osnovni izgled testa kot se uporablja sedaj. Vsaka `describe` funkcija vsebuje več `it` delov, ki se potem izvedejo zaporedno. Testi se poganjajo z ukazom v ukaznem oknu, kjer se prikaže tudi izpis rezultatov testiranja.

Poglavje 3

Razvoj domensko-specifičnega jezika

Cilj diplomske naloge je razviti domensko-specifični jezik, ki bo na ciljni platformi omogočal lažje ter predvsem hitrejše pisanje testov. Kako uspešni bomo pri tem, bomo presodili na koncu z analizo nastalega jezika, kjer bomo sedanje teste primerjali s tistimi, ki bodo napisani z novonastalim jezikom.

Trenutni sistem za izvajanje testov, je zaledni sistem za izposojo avtomobilov, ki je napisan v Node.js okolju. Za testiranje se trenutno uporablja Mocha okolje, ki ima posebno sintakso, ki bi jo tudi lahko opedelili kot domensko-specifični jezik. Ideja je razviti DSL, ki bo nadgradil oziroma nadomestil tovrstne teste, predvsem bomo poizkusili odstraniti vse dele testov, ki jih lahko generiramo avtomatsko. Tako bomo programerju prihranili čas, prav tako pa bo koda postala bolj pregledna.

DSL-ji se v testih učinkovito uporabljajo že sedaj, a so za naš vidik testiranja preobsežni, zato želimo zgraditi novega, ki se bo povsem osredotočil na naše potrebe. Naš namen je omogočiti učinkovito testiranje katerekoli aplikacije v tej tehnologiji. Testiranje z uporabo DSL-ja bomo razvijali agilno, pri čemer se bomo najprej osredotočili na bistvene funkcionalnosti. Morebitna nadgradnja bo prepuščena za poznejši čas.

Primerjava tehnologij Ruby in Node.js		
	Node.js	Ruby
Znanje jezika	veliko	majhno
Primeri razvoja DSL	malo primerov	veliko primerov
Sintaksa	jezik nam je znan	neznan jezik
Implementacija	enostavna	zapletena
Učenje jezika	hitro	počasno
Hitrost razvoja DSL-ja	hitro	počasnejše zaradi nepoznavanja jezika

Tabela 3.1: Primerjava tehnologij Ruby in Node.js

3.1 Izbira tehnologije

Prva odločitev je bila, da bomo razvijali **notranji domensko-specifični jezik**. Razlog izbire je bil zelo preprost. Teste smo želeli še naprej poganjati v Node.js okolju s pomočjo Mocha testov, saj nismo želeli izgubljati časa z implementiranjem nove tehnologije v ciljni projekt. Poleg že pridobljenega znanj v Node.js tehnologiji je vgradnja tega pristopa v sam projekt veliko lažja in bi ga lahko uporabili kot nekakšen paket. Pozitivna plat te izbire je tudi ta, da se bomo novega jezika zaradi znane sintakse naučili hitreje, kar znatno pospeši sam razvoj končnega izdelka - testov z novim jezikom.

Pri izbiri tehnologije, smo se odločali med dvema znanima tehnologijama: Ruby[18] in Node.js. V tabeli 3.1 lahko vidimo osnovno primerjavo med obema tehnologijama. Kot lahko razberemo iz tabele je edina velika prednost tehnologije Ruby ta, da obstaja veliko primerov razvoja domensko-specifičnih jezikov, ki so opisani za jezik Ruby on Rails[18], torej je znana in pogosta izbira za razvoj tovrstnih DSL-jev. Vseeno bi v našem primeru izgubili preveč časa za učenje jezika ter vgradnjo jezika v cilno platformo. Zato smo se na koncu odločili za tehnologijo Node.js.

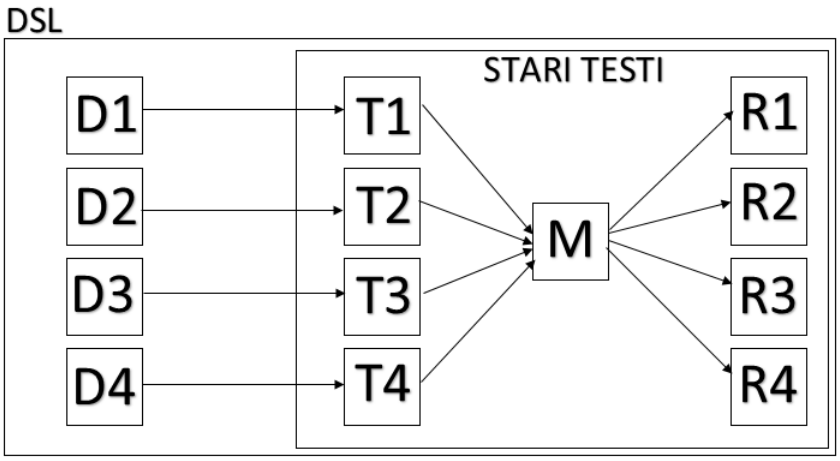
3.2 Zahteve za razvoj jezika

Pred razvojem domensko-specifičnega jezika smo skušali formalno definirati zahteve, ki se jih moramo držati oziroma jih mora jezik vsebovati.

- **Tehnologija**. Pri izdelavi DSL-ja smo vezani na izbrano tehnologijo Node.js. Z ravojem DSL-ja želimo omogočiti splošno rešitev znotraj izbranega okolja.
- **Odstranitev nepotrebne kode**. To je prvi in glavni cilj, ki ga želimo doseči. Testi vsebujejo veliko vrstic kode, ki jih lahko z novim jezikom generiramo avtomatsko in tako testerju olajšamo pisanje testov. Ker bomo jezik preverjali na projektu, s katerim imamo neposredne izkušnje, zelo dobro poznamo teste in posledično tudi lažje analiziramo strukturo testov. Hitro lahko ugotovimo, da imajo testi približno 70% kode, ki se ponavlja - veliko večino tega želimo odstraniti.
- **Hitrejši razvoj testov**. Celotno pisanje testov želimo pohitriti. To se pretežno nanaša na prvo zahtevo, saj v primeru, ko odstranimo nepotrebno kodo to posledično pomeni tudi to, da bomo teste pisali hitreje in z manj napakami.
- **Enostavnejša in pregledna koda**. Kodo želimo narediti bolj enostavno in pregledno, da bo ta tudi za zunanje opazovalce, vsaj približno jasna. Tudi ta del se navezuje na prvo zahtevo.
- **Enostavno poganjanje testov**. Teste bodo poganjali tudi uporabniki, ki niso vešči programeranja. V ta namen je potrebno zagotoviti, da se celotno testiranje sproži z enim samim ukazom.
- **Zanesljivost**. Ker domensko-specifični jeziki obravnavajo probleme na način, ki ni odvisen od implementacijskih podrobnosti, želimo izboljšati lovljenje napak pri izvajanju, kar nam bo omogočilo, da bodo testi vračali bolje opisane napake. Izboljšali bomo tudi validacijo in verifikacijo napak.

- **Možnost nadgradnje**. Celoten jezik želimo zastaviti ter razviti tako, da je vsako njegovo nadgradnjo možno narediti brez velikih težav. To je ena najpomembnejših zahtev, saj se testirani sistem hitro spreminja in nadgrajuje. Ker je potrebno teste dodajati ali popravljati dnevno, bo jezik kmalu potreben manjših popravkov.
- **Sodelovanje z razvijalci ciljnega sistema**. Ob razvoju jezika želimo aktivno sodelovati z razvijalci sistema. Tako želimo jezik čimbolj približati končnim željam razvijalcev. Uporabiti želimo agilen princip razvoja, saj se ta uporablja pri razvoju ciljnega projekta.

3.3 Delovanje DSL-ja



Slika 3.1: Potek testiranja z uporabo DSL-ja

Na sliki 3.1 je prikazna izvedba testov z uporavo novega DSL-ja. Trenutni testi se izvajajo v notranjem kvadratu: posamezni testi (T1,T2,...) se izvedejo preko Mocha okolja, na koncu pa izpišejo rezultate (R1, R2, ...), ki ustrezajo posameznemu testu.

Cilj diplomske naloge je, da teste izpišemo v kompaktnejši obliki s pomočjo DSL-ja (D1, D2, ...) Ti se potem pretvorijo v Mocha teste (T1, T2, ...) kjer se celoten postopek izvede enako kot se je doslej.

3.4 Razvoj DSL-ja

V naslednjih poglavjih bomo predstavili glavne korake, ki smo jih opravili pri razvoju jezika:

- analizo trenutnih testov,
- določitev osnovnega načrta jezika,
- tvorbo osnovnih funkcij,
- poganjanje testov in
- izpis rezultatov.

Pri vsakem koraku bomo predstavili glavne naloge ter težave, s katerimi smo se srečevali. Poleg tega bomo predstavili zglede pomembnih delov kode ter sproti obrazložili delovanje novega DSL-ja.

3.4.1 Analiza trenutnih testov

Prvi korak pri razvoju jezika je bil ta, da smo analizirali trenutno kodo testov in jo čimbolj razčlenili. Tako lahko pri načrtovanju domensko-specifičnega jezika lažje definiramo strukturo jezika ter odstranimo vso ponavljajočo se kodo.

Kot primer analize jezika si najprej poglejmo primer Mocha testa na kodi 3.1. Takoj opazimo, da se velik del kode ponavlja oziroma je že vnaprej definiran. Gre predvsem za ključne besede in primerjavo rezultatov, ki jih opravljamo z operacijo `assert`. V testih lahko prepoznamo določene vrstice, ki jih želimo (v podobni obliki) obdržati tudi v novem jeziku:

- **Ime testa**. Določitev imena testa moramo podati tudi v nobih testih, saj je to nujen pogoj za identifikacijo testa.
- **Metoda zahtevka ter ciljni naslov**. Posamezni testi se med seboj razlikujejo po metodi zahtevka. V naših testih uporabljamo na-

slednje metode: `GET`, `POST`, `PUT`, `DELETE`. Poleg tega vsak test uporablja drugačen ciljni naslov, saj le tako lahko vsako zahtevo ustrezno izvedemo.

- **Nastavitveni parametri**. Parametri se lahko razlikujejo od testa do testa in to je tudi spremenljiv del testov. Podrobnosti testa bomo podrobneje opisali v nadaljevanju.

- **Primerjava rezultata**. Za primerjavo rezultata uporabljamo `assert` funkcijo [19]. Celotna koda pa se zelo ponavlja, torej lahko tudi tu prihranimo veliko pisanja.

```
1  it('should return Slovenia', (done) => {
2        request(app)
3          .get(`${host}?filters={"code":["SI"]}`)
4          .set('Accept', 'application/json')
5          .end((err, res) => {
6            if (err) {
7              logger.error('Error occurred while test
                   getCountries(SI): %j', err);
8              return callback(err);
9            }
10           assert.equal(res.body[0].code, 'SI');
11           assert.equal(res.body[0].name, 'Slovenia');
12           assert.equal(res.body[0].currency, 'EUR');
13           assert.equal(res.body[0].cdpID, 'ONL-S01');
14           assert.equal(res.body[0].taxRate, '22');
15           done();
16         });
17     });
```

Koda 3.1: Primer GET testa

3.4.2 Osnovna sestava novih testov

V analizi testov smo definirali elemente, ki jih mora jezik obvezno vsebovati:

- **Objekt testa**. Jezik mora vsebovati konstruktor, ki sprejme dva parametra. Prvi parameter podaja lokacijo zalednega sistema, ki ga jezik avtomatsko požene na začetku testiranja. To je zelo pomembno, saj v nasprotnem primeru klici na določen URL naslov ne bi bili uspešni (recimo v primeru, ko sistem ne bi bil pognan). Po končanih testih sistem tudi prekine izvajanje. Drugi parameter v objektu je ime datoteke, v katero se bodo generirali testi. Tako bomo lahko pregledali končne rezultate - teste, naredili njihovo analizo ter primerjali rezultate s prejšnjimi testi.

- **Spreminjanje URL naslova**. Kot smo omenili se ponavlja tudi naslov, na katerega naredimo zahtevo. Določitev naslova želimo poenostaviti, zato pred posamezno skupino testov to izvedemo le enkrat ter tako pridobimo na hitrosti pisanja.

- **Razdelitev testov v skupine**. V Mocha okolju določimo skupino testov z pomočjo funkcije `describe`. Funkciji podamo ime skupine, v nadaljevanju pa ji naštejemo še vse teste, ki spadajo v to skupino. S poenostavitvijo lahko tudi tu pohitrimo razvoj testov.

- **Osnovne funkcije testov**. Osnovne funkcije lahko glede na metode zahtevkov `GET, POST, PUT, DELETE` razdelimo na 4 dele,. To bodo tudi naše glavne "funkcije". Pri pisanju želimo ohraniti predvsem tiste podatke, ki se spreminjajo. Tako bodo funkcije (odvisno od potrebe) prejemale naslednje podatke:

  - ime testa,
  - morebitne dodatne parametre k URL naslovu,
  - nastavitvene parametre, ki se zapišejo v glavo zahtevka,
  - podatke, ki jih pošljemo v glavnem delu zahtevka in

– objekte, v katere shranimo pričakovane rezultate.

- **Končna funkcija**. To funkcijo smo dodali z namenom, da izvede vse napisane teste. Teste se poganja zaporedoma, torej se testi poženejo po posameznih skupinah šele takrat, ko se zaključi izvajanje testov prejšnje skupine. Celotno testiranje tako postane bolj pregledno in lažje opazimo napake.

3.4.3 Tvorba osnovnih funkcij

Sledi opis razvoja posameznih delov ter predstavitev njihovega delovanja.

```
1  const makeSetObject = function(setObject, cb) {
2    let setData = `.set('Accept', 'application/json')`;
3    if (setObject === null) {
4      cb(setData);
5    } else {
6      async.each(Object.keys(setObject), (key, callback) => {
7          setData += `
8          .set('${key}', '${setObject[key]}')\n`;
9        callback();
10     }, () => {
11       setData = setData.substring(0, setData.length - 1);
12       cb(setData);
13     });
14   }};
```

Koda 3.2: Funkcija za obdelavo nastavitvenih parametrov

Nastavitveni parametri

Vsak zahtevek lahko vsebuje nastavitvene parametre, ki jih podamo v obliki JavaScript objekta. Ime spremenljivke znotraj objekta je parameter, ki ga želimo vstaviti, vrednost spremenljivke pa je vrednost parametra, ki ga pošljemo v zahtevku. Tako na enostaven način zmanjšamo velikost kode. Po-

leg tega so nekateri parametri dodani že v osnovi, saj so le tej nujno potrebni. Vse to je razvidno iz kode 3.2.

Primerjava rezultatov

```
1  const makeAssertObject = function(assertObject, cb) {
2    let assertData = '';
3    let arrayPosition = '';
4    if ('arrayPosition' in assertObject) {
5      arrayPosition = `[${assertObject.arrayPosition}]`;
6      delete assertObject.arrayPosition;
7    }
8    async.each(Object.keys(assertObject), (key, callback) => {
9      if(key === 'statusCode') {
10       assertData+=`assert.equal(res.${key}, ${assertObject[
             key]});\n`;
11     } else if (key === 'compareBody') {
12       assertData += `assert.ok(_.isEqual(res.body, ${
             assertObject[key]}));`
13     } else {
14       assertData += `assert.equal(res.body${arrayPosition}.${
             key}, '${assertObject[key]}');\n`;
15     }
16     callback();
17   }, () => {
18     assertData=assertData.substring(0, assertData.length-1);
19     cb(assertData);
20   });
21 };
```

Koda 3.3: Funkcija za primerjavo rezultatov

Omenili smo, da rezultate primerjamo s pomočjo funkcije `assert`. Zato podobno kot v prejšnji funkciji tudi tu pošljemo v funkcijo objekt, ki je strukturiran na podoben način. Razlika v tem primeru je, da lahko v objekt vstavimo nekaj vnaprej določenih ključnih besed in njihovih vrednosti, katere so namenjene za točno določeno nalogo. V novem jeziku omogočamo naslednje dodatne možnosti:

- `arrayPosition` pove, kateri rezultat v prejetem objektu moramo primerjati.
- `statusCode`,preveri kodo odgovora ter tako določi, če smo prejeli pričakovan odgovor.
- `compareBody` omogoča primerjavo celotenega rezultata ter tako do potankosti preveri, če smo prejeli ustrezen odgovor.

Ta del je predstavljen na primeru kode 3.3.

GET zahteva

V prejšnjem razdelku smo si ogledali *GET* zahtevo, sedaj pa si poglejmo še to, kako smo sprogramirali obdelavo tega dela. Pri *GET* metodi je posebnost ta, da v zahtevku ne pošljemo podatkov v glavnem delu in ta sprejme le štiri parametre.

```
1  Test.prototype.get=function(itName,host,setObject,assertObj){
2    host = host || '';
3    makeAssertObject(assertObj, (assertData) => {
4      makeSetObject(setObject, (setData) => {
5        this.data += `
6          it('${itName}', (done) => {
7            request(app)
8              .get('${this.host}${host}')
9              ${setData}
10             .end((err, res) => {
11               try {
12                 ${assertData}
13                 done();
14               } catch (e) {
15                 done(e);
16               }
17             });
18         });`;
19     }); });};
```

Koda 3.4: GET funkcija

Posledica tega je, da pisanje testov lahko zelo poenostavimo. Kot je razvidno iz vrstice 5 v kodi 3.4, se celoten test generira avtomatsko, doda se le nekaj globalnih spremenljivk ter argumenti, ki jih funkcija prejme. Preden argumente dodamo v test, jih pošljemo še v obdelavo v druge funkcije.

POST zahteva

POST metoda za razliko od *GET* metode prejme tudi podatke, ki jih pošljemo v glavnem delu zahtevka. Ponavadi se to uporablja zato, da željeni objekt dodamo v bazo podatkov. Poleg tega se metoda ne razlikuje po ničemer. Primer lahko vidimo v kodi 3.5.

```
1  Test.prototype.post=function(itName,host,setObj,data,
       assertObj){
2    host = host || '';
3    makeAssertObject(assertObj, (assertData) => {
4      makeSetObject(setObj, (setData) => {
5        this.data += `
6          it('${itName}', (done) => {
7            request(app)
8              .post('${this.host}${host}')
9              ${setData}
10             .send(${JSON.stringify(data)})
11             .end((err, res) => {
12               try {
13                 ${assertData}
14                 done();
15               } catch (e) {
16                 done(e);
17               }
18             });
19         });`;
20     });
21   });
22 };
```

Koda 3.5: POST funkcija

3.4.4 Poganjanje testov

V zahtevah smo omenili, da bi radi vse teste pognali z enim samim ukazom v ukaznem oknu. Med iskanjem ustrezne rešitve smo se odločili, da bomo uporabili funkcijo, ki jo omogoča Node.js in njegov urejevalnik paketov `npm`. Ta je enostavna in prijazna za uporabo, saj v ciljnem projektu v datoteko `package.json` vnesemo le vrstice kode na sliki 3.6

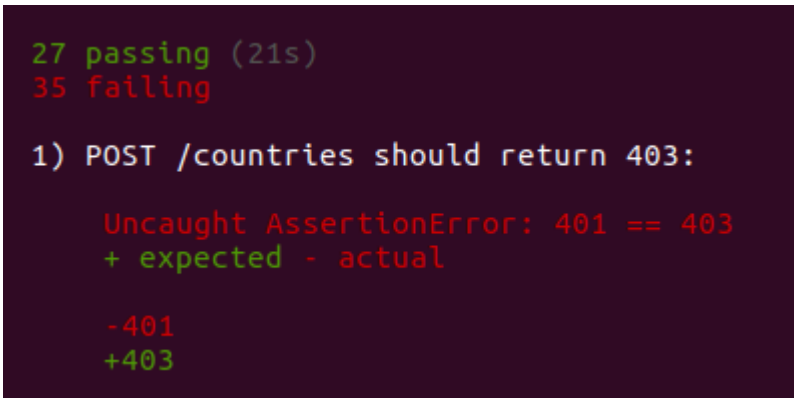
```
1    "test": "node index.js && mocha test.js"
```

Koda 3.6: npm funkcija

Priložena koda se potem preko ukaznega okna izvede z ukazom `npm run test`. Ukaz najprej požene ukaz na prvi datoteki, ki s pomočjo novega jezika ustvari datoteko, v kateri so osnovni testi. Na teh se potem požene ukaz za poganjanje testov `npm run test`.

3.4.5 Izpis rezultatov

Pri izpisu rezultatov uporabljamo standarden izpis Mocha okolja, saj se nam zdi ta najbolj primeren, Prav tako nam ta posreduje veliko podatkov v primeru, ko test ne uspe. Primera izpisa sta prikazana na slikah 3.2 in 3.3.



Slika 3.2: Izpisa testov, ki prikaže število uspešnih in neuspešnih testov

```
5) PUTCountries should return error status 403:

    AssertionError: 201 == 403
    + expected - actual

    -201
    +403

    at Test.request.put.set.set.send.end.e (test.js:245:20)
    at Test.assert (node_modules/supertest/lib/test.js:179:6)
    at Server.assert (node_modules/supertest/lib/test.js:131:12)
    at emitCloseNT (net.js:1555:8)
    at _combinedTickCallback (internal/process/next_tick.js:71:11)
    at process._tickCallback (internal/process/next_tick.js:98:9)

6) PUTCountries should return error status 404:

    AssertionError: 500 == 404
    + expected - actual

    -500
    +404

    at Test.request.put.set.set.send.end.e (test.js:267:20)
    at Test.assert (node_modules/supertest/lib/test.js:179:6)
    at Server.assert (node_modules/supertest/lib/test.js:131:12)
    at emitCloseNT (net.js:1555:8)
    at _combinedTickCallback (internal/process/next_tick.js:71:11)
    at process._tickCallback (internal/process/next_tick.js:98:9)
```

Slika 3.3: Primer izpisa testov, ko se test izvede neuspešno

Poglavje 4

Pisanje testov

V nadaljevanju si bomo ogledali testiranje z uporabo novega DSL-ja. Ta določa splošno obliko testov, ki se potem od primera do primera razlikuje. Prikazane so vse tipišne oblike testov, ki jih DSL v trenutni obliki omogoča.

4.1 Prikaz oblike novih testov

V kodi 4.1 je predstavljena oblika novih testov. Vsi rdeče izpisani deli so v testih ustezno nastavljeni. Pri tem velja:

```
1 test.METODA('IME TESTA', 'URL', {
2   NASTAVITVENI PARAMETRI }, PODATKI, {
3   REZULTATI
4 });
```

Koda 4.1: Primer obliike novih testov.

- `METODA`. Tu izbiremo metodo, s katero naredimo zahtevo na ciljno platformo. Možnosti so: `GET, POST, PUT, PATCH, DELETE`.
- `IME TESTA`. Vsem testom je potrebno določiti ime, saj nam to omogoča lažje razhroščevanje.
- `URL`. V URL podamo cilj, na katerega se izvede zahteva.

- `NASTAVITVENI PARAMETRI`. V primeru, da je potrebno nastaviti dodatne parametre pri zahtevku, le-te podamo v obliki objekta.
- `PODATKI`. Podatke, ki jih želimo poslati na ciljno platformo, podamo v obliki objekta. Pri `GET in DELETE` metodah tega podatka ne nastavimo.
- `REZULTATI`. Rezultate, za primerjavo podamo v obliki objekta kot zadnji parameter v funkciji.

4.2 Priprava okolja

Ob začetku testiranja je potrebno najprej pognati okolje. Prvo opravilo je vpeljava vseh potrebnih spremenljivk. Prav tako moramo narediti novo instanco testov, katerim podamo pot do našega zalednega sistema,ter ime datoteke, v katero se testi generirajo. Enostaven primer lahko vidimo na primeru kode 4.2. Uporabljamo okoljske spremenjivke (angl. environment variables) [20], saj do njih lahko dostopamo preko vseh procesov in na različnih sistemih, brez spreminjanja kode. Okoljske spremenljivke so v večini primerov shranjene v začasnih datotekah, ki se nahajajo v domačem imeniku uporabnika.

```
1  const Test = require('./index');
2  var test = new Test('./Mobility/service/server', 'test.js');
3  const userToken = process.env.CR_USER_TOKEN;
4  const adminToken = process.env.CR_ADMIN_TOKEN;
5  const countryData = {
6    code: 'ITA',
7    name: 'Italia',
8    currency: 'EUR',
9    cdpID: 'CODE-ITA'
10 };
```

Koda 4.2: Priprava okolja

4.3 Testiranje URL naslova

Ker testiramo različne URL naslove, ki vračajo različne odgovore, si bomo na primeru pogledali le izbrani del testov.

4.3.1 Definiranje URL-ja skupine testov

Pred začetkom pisanja testov za določeno skupino je potrebno definirati URL, na katerega bomo pošiljali zahtevke. Določimo tudi ime skupine testov ter najdaljši dovoljeni čas izvajanja testov v milisekundah. V primeru, da se test ne izvede v tem času, se testiranje prekine z negativnim rezultatom. Primer vidi v kodi 4.3.

```
1 test.setHost('/api/countries');
2 test.start('GET Countries', 4000);
```

Koda 4.3: Definiranje skupine testov

4.3.2 Primer GET zahtevka

Po implementaciji `GET` metode, si oglejmo še njeno uporabo. Metodi podamo ime testa, dodatek k URL naslovu, ter nastavitvene parametre, ki so v tem primeru NULL. Na koncu dodamo še objekt za preverjanje rezultata in ključne besede. Vse ostalo so polja, ki jih želimo preveriti. Stare teste s kode 3.1 smo tako v kodi 4.4 zmanjšali za polovico.

```
1 test.get('should return Slovenia', '?filters={"code":["SI"]}'
    , null, {
2   arrayPosition: 0,
3   code: 'SI',
4   name: 'Slovenia',
5   currency: 'EUR',
6   cdpID: 'ONL-S01',
7   statusCode: 200,
8 });
```

Koda 4.4: Primer novega GET testa

4.3.3 Primer POST zahtevka

Pri `POST` metodi funkciji podamo podobne parametre kot pri GET-u, vendar tu (lahko) podamo tudi nastavitvene parametre - recimo ključ za dostop do sistema. Podamo tudi podatke, ki se pošljejo na zaledni sistem. V priloženem primeru kode 4.5 testiramo primer, ko nam zaledni sistem podatka ne dovoli zapisati v bazo. Podoben test brez uporabe novega pristopa potrebuje kodo, ki ima trikrat več vrstic.

```
1 test.post('should return error - duplicate key', '', {
2   Authorization: adminToken,
3 }, countryData, {
4   errorCode: 1003,
5   statusCode: 409,
6   message: 'Error while managing record.',
7 });
```

Koda 4.5: Primer novega POST testa

4.3.4 Primer AFTER funkcije

V jezik smo implementirali tudi nekaj dodatnih funkcij, ki olajšajo izvedbo testov: `After, Before, AfterEach, BeforeEach`. Te funkcije se izvedejo pred začetkom ali po koncu izvajanja testov v posamezni skupini, oziroma se izvedejo tudi pred in po vsakem testu v skupini. So zelo pomembne, saj tako odstranimo posledice predhodnih testov (recimo povrnemo bazo podatkov v enako stanje kot je bila na začetku). Primer v kodi 4.6 prikazuje, kako po končanih testih odstranimo vnos v bazo podatkov, ki smo ga naredili med izvajanjem testov.

```
1 test.after('remove country', (done) => {
2   const Country = require('./models/countries/countries');
3   Country.findOneAndRemove({ code: 'ITA' },()=>{ done(); });
4 });
```

Koda 4.6: Primer After funkcije

4.4 Zaključek skupine testov ter poganjanje

Test zaključimo ter poženemo samo z dvema vrsticama kode (koda 4.7). Prva je namenjena zaključku skupine testov, druga pa generiranju splošne funkcije, ki bo kasneje izvajala posamezne teste zaporedno. Na ta način med drugim olajšamo tudi razhroščevanje napak.

```
1 test.end();
2 test.runTests();
```

Koda 4.7: Primer zaključka testov.

Teste na koncu poženemo s pomočjo ukaza, ki ga izvedemo v ukaznem oknu (koda 4.8). Z njegovo pomočjo jezik generira teste, ki se poženejo ter izvedejo na ciljnem sistemu. Ko se testi izvedejo, nam program izpiše rezultat.

```
1 npm run test
```

Koda 4.8: Ukaz za začetek testiranja.

4.5 Testiranje delovanja jezika

Na koncu smo jezik seveda tudi testirali, če deluje tako, kot se je pričakovalo. Testiranje smo izvedli na pilotnem projektu za izposojo avtomobilov. Nov način smo preverjali trije testerji, ki smo poleg tega tudi programerji na pilotnem projektu. Napisali smo 72 testov, ki so testirali različne naslove in uporabljali različne metode. Testiranje smo izvajali ob spremembi zalednega sistema, kar pomeni, da je bilo potrebno nekatere teste tudi dodati ali pa spremeniti.

Pri testiranju se je izkazalo, da je povprečen čas izvajanja testov primerljiv z izvajanjem prejšnjih testov. Kot velika prednost se je izkazalo predvsem to, da jezik morebitne napake sporoča bolj pregledno. To je tudi omogočilo lažje razhroščevanje napak v zalednem sistemu.

4.6 Rezultati dela

V tem delu si bomo pogledali primerjavo testov, ki so napisani z našim jezikom ter testov, ki smo jih napisali pred tem.

Delovanje novega jezika smo testirali na pilotnem projektu. Projekt raz-

Primerjava testov		
	novi testi	stari testi
število vrstic	489	1487
število znakov	12187	45887
število testov	72	72
čas izvajanja testov	64s	66s
čas razvoja testov	8h	40h

Tabela 4.1: Primerjava testov

vijamo leto dni in je še vedno v aktivnem razvoju. Trenutno je na projektu zaposlenih 5 programerjev, ki so poleg tega tudi testerji. Ker je projekt postal preobsežen za ročno testiranje, smo se odločili, da vpeljemo avtomatsko testiranje. To smo rešili z okoljem Mocha, stanje testov pred razvojem novega jezika pa si lahko ogledamo v tabeli 4.1. Omenimo naj, da je kot čas razvoja testov štet tisti čas, ki ga namenimo pisanju novih testov ne pa tudi popravljanju le-teh.

Po razvoju testov za pilotni projekt z novo rešitvijo smo naredili kratko primerjavo z starimi testi. Kot je prikazano na tabeli 4.1, smo število vrstic zmanjšali za *trikrat*, število znakov pa kar za *štirikrat*. Število testov je pri tem ostalo enako, saj smo le tako lahko naredili neposredno primerjavo dveh različnih rešitev. Čas izvajanja novih testov je bil daljši, vendar v povprečju le za nekaj sekund, kar je pri tako velikem projektu zanemarljivo. Sam razvoj testov je postal občutno hitrejši, saj smo ta del pospešili za *petkrat*. Ponovno naj omenimo, da v ta čas ni všteto popravljanje testov.

Med uporabo novonastalega jezika smo prejemali tudi odziv uporabnikov našteli pa bomo nekaj najbolj pomembnih:

- sintaksa testov je zelo poenostavljena,
- razvoj testov je hitrejši ter lažji,
- izpis rezultatov je pregleden in razumljiv,
- jezik je lahek za učenje in zelo razumljiv,
- implementacija jezika v projekt je enostavna,
- zaključek testov ter poganjanje testov bi lahko naredili avtomatsko brez dodatnih funkcij,
- iz kode bi odstranili začeten ukaz `test` ter tako teste začeli z imenom funkcije (`GEt, POST, PUT, DELETE,...`).

Poglavje 5

Sklepne ugotovitve

5.1 Analiza

Z uporabo našega DSL-ja se je čas pisanja testov skrajšal za več kot petkrat, velikost kode pa se je zmanjšala kar za tretjino. Med uporabo novega jezika smo opazili tudi opazno zmanjšanje števila napak - predvsem po zaslugi lažje berljive kode in krajše sintakse testov.

Z vidika uporabnika je razvoj testov z uporabo DSL-ja hitrejši ter enostavnejši. Tu moramo upoštevati, da je razvoj DSL-ja sicer trajal kar nekaj časa, vendar bomo ob njegovi nadaljni uporabi teste pisali hitreje, kar nam bo omogočilo hitrejši razvoj ciljne rešitve. Omenimo naj tudi to, da je implementacija testov v okolje enostavna, poenostavljeno pa je tudi samo izvajanje testov preko ukaznega okna.

Ker smo se ob razvoju jezika odločili za izdelavo notranjega DSL-ja, je bilo potrebno med razvojem jezika sprejeti nekaj kompromisov, saj se je bilo potrebno držati sintakse, ki jo javascript ponuja.

Končno mnenje avtorja diplomskega dela je, da smo z razvojem domensko-specifičnega jezika olajšali delo tako razvijalcem kot testerjem. Jezik naj bi se v prihodnosti uporabljal tudi na drugih projektih ter pripomogel k boljši končni rešitvi. Ključno za uspeh je bilo, da smo jezik enostavno in hitro implementirali v okolje zalednega sistema ter tako omogočili hitro uporabo

v praksi.

5.2 Nadaljni razvoj

Za konec naj omenimo še načrte za nadgradnjo jezika v prihodnosti.

- Testiranje želimo opraviti tudi na ostalih sistemih, da bi še bolje preverili njegovo delovanje. Tako bi pokazali, da jezik že lahko uporabimo na drugih rešitvah.
- Med samim razvojem jezika se je pojavila ideja, da bi uporabili nastavitveno datoteko, kjer bi bolj podrobno upravljali z nastavitvami jezika.
- Vpeljati je treba tudi dodatne funkcije, ki smo jih v dosedanjem razvoju izpustili.
- Izpis rezultatov testiranja bi lahko naredili še bolj pregleden, poleg tega pa bi rezultate lahko zapisali še v datoteko.
- Vpeljali bi lahko orodje, ki bi s testi preverjalo pokritost kode na ciljnem projektu. Tako bi lahko hitro opazili, če potrebujemo nove teste, ki se morajo izvesti na nekem delu kode, ki jo trenutni testi ne pokrivajo.
- Jezik bi ob morebitni razširitvi lahko objavili tudi kot odprtokoden projekt in ga v okolje Node.js vmestili kot paket.

Literatura

[1] Javascript. https://en.wikipedia.org/wiki/JavaScript. [Dostopano: 22. 5. 2017].

[2] Node.js. https://nodejs.org/en/. [Dostopano: 22. 5. 2017].

[3] Mocha. https://mochajs.org/. [Dostopano: 22. 5. 2017].

[4] Domain-specific language. https://en.wikipedia.org/wiki/Domain-specific_language. [Dostopano: 20. 5. 2017].

[5] Html. https://www.w3schools.com/html/. [Dostopano: 10. 6. 2017].

[6] Latex. https://www.latex-project.org/. [Dostopano: 10. 6. 2017].

[7] M. Mernik, A. M. Sloane, J. Heering. When and how to develop domain-specific languages. v zborniku: ACM Computing Surveys, 37(4):316–344, 2005.

[8] Definicija domensko-specifičnega jezika. https://www.quora.com/What-is-a-DSL-domain-specific-language-How-does-it-differ-from-the-other-programming-languages. [Dostopano: 16. 6. 2017].

[9] Splošno namenski jeziki. http://wiki.c2.com/?GeneralPurpose\discretionary{-}{}{}ProgrammingLanguage. [Dostopano: 20. 5. 2017].

[10] Programski jezik. https://sl.wikipedia.org/wiki/Programski_jezik. [Dostopano: 18. 5. 2017].

[11] Markus Voelter. Dsl engineering: Designing, implementing and using domain-specific languages. http://voelter.de/data/books/markusvoelter-dslengineering-1.0.pdf, 2013.

[12] Martin Fowler. Domain-specific languages. http://ptgmedia.pearsoncmg.com/images/9780321712943/samplepages/0321712943.pdf, 2010.

[13] Java. https://www.java.com/en/. [Dostopano: 16. 6. 2017].

[14] Steven Kelly, Juha-Pekka Tolvanen. *Domain-Specific Modeling Enabling Full Code Generation*. 2008.

[15] Martin Fowler. Language workbenches: The killer-app for domain specific languages? https://www.martinfowler.com/articles/languageWorkbench.html, 2005.

[16] Semantika programskega jezika. https://en.wikipedia.org/wiki/Semantics_(computer_science). [Dostopano: 16. 6. 2017].

[17] npm. https://www.npmjs.com/. [Dostopano: 22. 5. 2017].

[18] Ruby. https://www.ruby-lang.org/en/. [Dostopano: 10. 6. 2017].

[19] Assert funkcije. https://nodejs.org/api/assert.html. [Dostopano: 16. 6. 2017].

[20] Okoljske spremenljivke. https://en.wikipedia.org/wiki/Environment_variable. [Dostopano: 10. 6. 2017].